Criticality-Based Optimizations for Efficient Load Processing

Samantika Subramaniam Anne Bracy[†] Hong Wang[†] Gabriel H. Loh

Georgia Institute of Technology
College of Computing
Atlanta, GA, USA
{samantik,loh}@cc.gatech.edu

[†]Intel Corporation
Microarchitecture Research Laboratory
Santa Clara, CA, USA
{anne.c.bracy,hong.wang}@intel.com

Abstract

Some instructions have more impact on processor performance than others. Identification of these critical *instructions can be used to modify and improve instruction processing. Previous work has shown that the criticality of instructions can be dynamically predicted with high accuracy, and that this information can be leveraged to optimize the performance of load value prediction and instruction steering for clustered architectures. In this work, we revisit the idea of criticality, but we propose several processor enhancements that can exploit criticality information and can be directly applied to modern x86 microarchitectures. For the investment of a small (less than 1KB) criticality predictor, we can make a conventional single-read-port data cache achieve the performance of an ideal dual-read-port cache, yielding an average 10% performance improvement. Our remaining techniques can reuse the predictor (i.e., no additional overhead) to further optimize other aspects of load processing (e.g., caching decisions, store-to-load forwarding, etc.), yielding an overall performance improvement of 16% over a conventional processor. Some of these techniques also allow us to decrease power and area costs for several related hardware structures.*

1. Introduction

The sensitivity of overall processor performance to the execution latencies of different instructions may vary for many reasons. For some instructions, any increase in the instruction's execution latency, even by a single cycle, will result in a corresponding increase in overall program execution time. Such instructions are said to be on the program's *critical path* of execution (or the instruction is called *critical*). For other instructions, one may be able to increase the latency of execution (or defer the start of execution) by one or more cycles without impacting performance.

One of the limitations of past work on exploiting the criticality of instructions is that the proposed applications are predominantly geared toward specialized microarchitectures. The first primary application area was for dynamic value prediction [7, 11, 31], which after years of research has not been adopted in a commercial processor. The other primary application area is for clustered microarchitectures with cross-cluster bypass penalties [11, 23, 31] or asymmetric power or performance characteristics [5, 10, 19, 24], neither of which will be used in any known future *main-stream* commercial processors. Other proposed applications include the Non-Critical Load Buffer [12] and the Penalty Buffer [4], both of which augment the DL1 cache with an auxiliary buffer that stores data accessed by non-critical loads. Although the proposed buffers are relatively small, adding any new structures to the DL1 processing path is complex as memory accesses must get routed to multiple structures, results require extra multiplexing, and it also represents one more structure that must be snooped for maintaining cache coherence.

So while research has shown that the criticality of instructions, and loads in particular, can be accurately predicted, the hardware applications of criticality prediction so far seem to be limited to more specialized or niche microarchitectures. In this work, we describe a simple but effective load criticality predictor, and then propose six new and simple criticality-based optimizations that can be directly applied to modern, conventional x86 microprocessor organizations. Our applications are divided into three categories based on where in the load's execution pipeline they are applied. The first class involves the data cache port of a processor where we use criticality predictions to make a single-load-port data cache perform almost as well as an ideal dual-load-port version. The second class deals with the memory disambiguation and store-to-load forwarding. The third class includes applications that modify the placement of data in the Level 1 data cache (DL1) to increase the hit-rates of critical loads. While the load-port optimization for the DL1 has the greatest performance benefit, once the overhead for the criticality predictor has been paid for, implementing any (or all) of the remaining optimizations are effectively "free." By simultaneously combining several of our criticality-based optimizations, not only is the performance benefit higher, but the cost of the hardware predictor also gets amortized across all of these multiple optimiza-

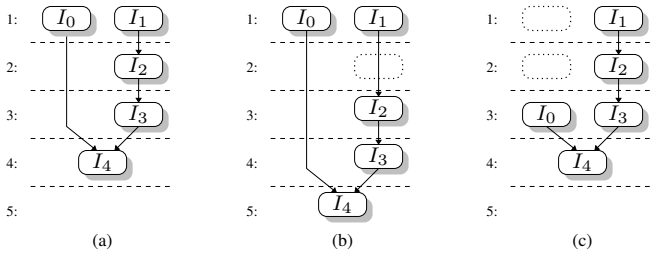

Figure 1: Dataflow graph that depicts how delaying the execution of some instructions increases the critical path.

tions.

The rest of the paper is organized as follows. Section 2 reviews the concept of load criticality and provides a brief overview of the most relevant prior research. Section 3 describes the design and operation of our proposed critical load predictor and Section 4 describes our simulation infrastructure. We then propose and evaluate several applications in Sections 5-7, and we consider the combination of these techniques in Section 8. We also compare our load criticality predictor with another previously proposed criticality predictor in this section. We draw some final conclusions in Section 9.

2. Background and Related Work

In this section we first define the criticality of instructions and explain the conditions that make some loads more critical than others. We then briefly describe some of the most relevant prior studies and highlight some important observations.

2.1. Instruction (and Load) Criticality

Criticality can be described with a program's data-dependency graph; there exists one or more paths through the graph with a maximum length; such a path is called a *critical path*. Any instruction that appears in this path (or paths) is critical. Consider the dataflow graph shown in Figure 1(a). If each instruction takes a single cycle to execute, then the longest path through this graph (the critical path) has a length of four, starting from I_1 and ending at I_4. Therefore, all four instructions I_1 to I_4 are critical instructions. For example, Figure 1(b) shows that when I_2's execution is delayed by a cycle, then length of the longest path increases. In contrast, instruction I_0 is not critical because we can delay its execution (or increase its execution latency) and the critical path length remains unchanged as shown in Figure 1(c). Note that at some point, if we delay I_0 by enough cycles (in this case more than two cycles), it can form a new longest path and therefore become critical.

Critical loads are simply load instructions that are critical. The reason we focus on loads is that they often have long latencies due to cache misses in the L1 or L2, and therefore are far more likely to lie on the critical path of execution. The criticality of loads can manifest in many ways. First, there is the simple case where the load is on the critical path in the traditional dataflow sense, as described earlier. Second, delaying loads that have a mispredicted branch as a child (or a grand-child, etc.) directly delays the detection and subsequent recovery from the branch misprediction. Third, when a load is the oldest instruction in the processor and suffers a long latency cache miss, the reorder buffer may become full and new instructions (regardless of whether they are data dependent on the load or not) will not be able to allocate resources and execute.

The observation that some instructions are critical while others are not has been known for quite some time [4, 28]. More recently, work has gone into carefully quantifying the properties of the criticality [27, 32] and slack of instructions [10], as well as the dynamic predictability of criticality [11, 31].

2.2. Predicting Instruction Criticality

In this sub-section, we focus on previous proposals to predict instruction criticality.

2.2.1. Dataflow-Based Prediction

Fields et al. proposed a token-passing algorithm that develops dependence chains and learns the critical path of the program [11]. The predictor builds dependence graphs based on last-arriving chains of instructions assuming that the critical path is likely to reside along the last-arriving edges. The corresponding hardware implementation uses an array that tracks certain tokens that are distributed randomly at different instruction points. A set of pre-decided rules builds a chain of last-arriving edges from the root instructions (where the tokens are planted) along which tokens are propagated. All tokens that are propagated are checked for "liveness" after a certain number of instructions. If a token is alive when checked, the instruction at which the token was inserted is on the path of a last arriving edge and is deemed to be critical. While very accurate due to its explicit tracking through the program's dataflow graph, this approach is difficult to implement in hardware due to the management of the token-based array, token free list, and the simultaneous tracking of multiple tokens.

2.2.2. Criticality Prediction from Implicit Criteria

Several research proposals for criticality predictors make use of information other than explicit dataflow to *infer* criticality in a more heuristic-based manner. Fisk and Bahar proposed two indicators of load criticality [12]. First, loads that cause the processor utilization to fall below a certain threshold are considered to be critical. The second heuristic considers the number of instructions that are added to the load's "dependency chain" over the course of a miss. To measure the number of dependents added during the course of a miss, the predictor needs to first track the number of dependencies at the time when a cache miss is detected, and

again when the miss returns, to check for additional dependencies. Their proposed implementation adds counters to the load queue and MSHRs, plus some extra state to track which counter should be updated. Particularly troublesome for implementation is the need to communicate or broadcast consumer information from the allocation stage to the MSHR entries behind the DL1 cache. One interesting aspect of these predictors is that a load miss acts as the trigger to start tracking the criticality of an instruction, thereby reducing the number of instructions that need to be tracked.

Tune et al. also proposed several criticality predictors based on simple heuristics [31]. The heuristics consider the age of the instructions, the number of dependents, and the number of dependents made ready. While the heuristics are easy to describe, they are not all easily implementable in hardware. For example, the "QOldDep" criteria marks each instruction in the scheduling queue (RS) that is dependent on the oldest instruction that is in the queue as critical. This requires first knowing the oldest instruction (which may change every cycle), and then somehow broadcasting the information to all other instructions so that they can be marked as critical if they are dependent. Another heuristic, "QCons" checks each completing instruction and marks the one whose result is used by the most instructions in the queue that cycle as critical. Implementing this in hardware would be expensive as it requires some way to immediately detect the number of physical register tag matches and then compare which instruction caused the most.

Srinivasan et al. proposed a more specific heuristic-based predictor that considered the instruction type of the dependents [27]. Their scheme proposes three conditions to classify a load as critical: 1) any of its dependents is a mispredicted branch, 2) any of its dependents is a load that misses in the cache, 3) the number of independent instructions issued (within a certain number of cycles) following this load is below a pre-determined criticality threshold. The predictor table used in this proposal is comprised of dependency vectors, one per ROB-entry which tracks the number and type of instructions waiting for a specific load. This dependence vector increases with both the ROB size and the LQ size and can be a considerable area overhead.

3. Our Load-Criticality Predictor

Many of the previously proposed criticality predictors, while effective, are fairly complex to implement directly in hardware without severely impacting one or more major functional unit blocks (e.g., the out-of-order scheduling logic). In this section, we describe our simple yet effective criticality predictor used in this study.

3.1. Basic Operational Description

As a base principle for our load criticality prediction algorithm, we use the observation made by Tune et al. and Fisk and Bahar that a load instruction with many dependents is likely to be on the critical path. A simple statistical explanation for this is that an instruction with many consumers has that many more chances for one of them to be on the critical path, thereby making the original parent more likely to be critical as well. For our algorithm, the prediction hardware includes two main components: a Consumer Collection Logic (CCL) and a Critical Load Prediction Table (CLPT).

Consumer Collection Logic The CCL is responsible for collecting the number of direct consumers (i.e., the load's children, but *not* its grand-children or other deeper descendants) that each load has while the load is in the instruction window. The CCL inspects every allocated instruction: if it is a load, the CCL sets a bit associated with the logical register that this load writes to, indicating that the instruction which wrote to this register is a load. This bit is concatenated to the physical register mapping stored in the register alias table (RAT). Additionally, the CCL resets a counter in the ROB entry corresponding to the load. If the instruction is not a load, then the CCL reads the input operand mappings from the RAT and determines if the parent is a load. Note that intra-group dependencies are also automatically handled by the existing intra-group dependency check and bypass logic, and we simply piggy-back on the regular RAT reads and writes so we do not introduce the need for any additional RAT ports. If the source is a load, we increment the counter associated with load's ROB entry.

Critical Load Prediction Table When a load instruction commits, the consumer count that is associated with its ROB entry is written to the CLPT, which is a simple, untagged, PC-indexed table of k-bit counters. For each load that we allocate, we use the load's PC to access the CLPT to determine the number of consumers it had the last time we saw the same load. Based on the consumer count associated with its entry, a load is predicted to be critical or not. For a single application of load criticality, each entry of the table would only need to have a bit to store whether the load is critical or not, which in turn would be determined by whether the number of consumers exceeded some pre-specified threshold. In this paper, we examine multiple applications of load criticality where each technique may have different criticality criteria (i.e., different thresholds). By storing the consumer count directly in the CLPT, we can reuse the same prediction structure across all of our optimizations, even if each optimization requires a different criticality threshold.

3.2. Example

The following example demonstrates how the CCL collects consumer information, and how the CLPT gets updated. Consider the four instructions listed in Figure 2. Even though the CCL is responsible for controlling all of the data collection, we do not explicitly show it in the figures for

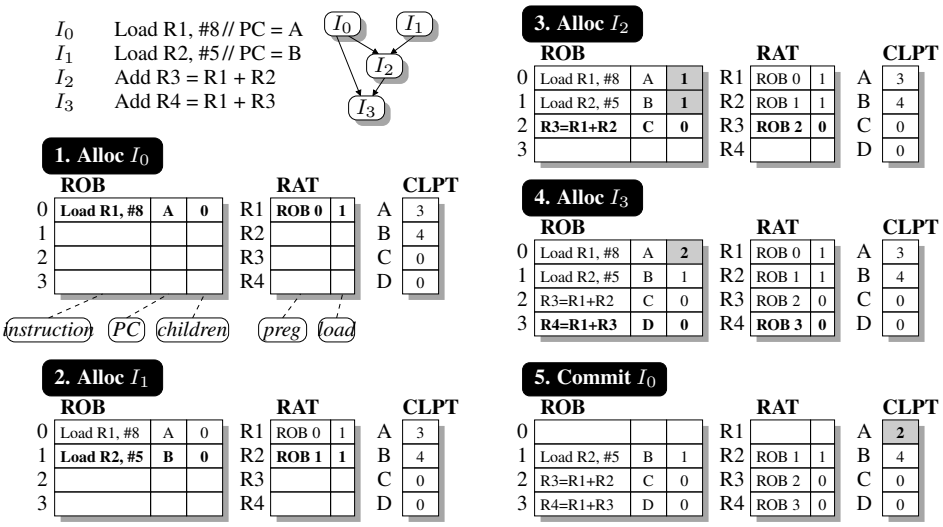

Figure 2: Example showing the collection of consumer counts and update to the prediction table.

simplicity. For each load, its ROB entry has a counter that tracks the number of immediate consumers. Every RAT entry has a load bit that indicates if the instruction that writes to the corresponding logical register is a load. Each snapshot corresponds to a specific operation on a single instruction. The example starts with the CLPT with some counter values that would have been collected the last time these loads were executed. All updates in a given step are highlighted with bold text.

1. Alloc I_0 In this step, the first load, I_0 is allocated. Since it is the first instruction, it is assigned ROB index 0. This ROB ID also serves as the renamed physical register (*preg*). The load bit is set in the RAT indicating that the instruction that wrote to register R1 is a load. The consumer counter in the ROB entry is set to 0. In parallel with this, the load PC A is used to look up the CLPT. The entry in the CLPT has a value of 3 indicating that the last time this load was seen, it had three consumers. Depending on the criticality threshold, this instruction may be marked as critical.

2. Alloc I_1 In this step, the second load, I_1 is allocated. It is assigned ROB ID 1, and the consumer counter in the ROB entry is set to 0.

3. Alloc I_2 In this step, the first add is allocated. First, the RAT entries that correspond to the source operands of the add are checked to see if either source is a load. For this add, the RAT entry for the left operand indicates that its parent is a load. Using the ROB ID stored in the RAT, we now increment the corresponding child counter in ROB entry 0 (shown shaded in the ROB). Similarly, the right parent is also a load, and so we also increment the respective counter in ROB entry 1 (also shaded).

4. Alloc I_3 In this step, the second add is allocated. Checking the RAT, we see that the left source is a load, and so the counter in ROB 0 is incremented. The right operand is present in the ROB, but since the load-bit in the RAT indicated that the right parent is not a load, no further actions are taken. This add instruction is inserted into the ROB at entry 3.

5. Commit I_0 In this step, load I_0 commits. We copy its counter value from the ROB into the corresponding CLPT entry to reflect its new consumer count which is 2. This new consumer count value will be used to predict the load criticality the next time Load A is encountered (or any other load that aliases to the same predictor entry).

3.3. Issue-Rate Filtering of the Predictor

In the preceding sub-sections, we described and demonstrated the basic operation of our load criticality predictor. While the number of in-flight consumers of a load is a strong indicator of its importance, the load's criticality also depends on the dynamic processor conditions while the load is in the instruction window. Consider a load with five direct consumers in the window. Such a load will be considered critical if the criticality threshold is four. If there are enough independent instructions in the window that can execute in parallel with this load however, then the latency of the load may not significantly impact the performance of the processor. We restrict our algorithm to only collect consumer counts when the processor is in a period of low processor utilization or a "critical period." The idea is that when the issue rate of the out-of-order processor is low, then the probability of finding one or more long-latency, critical loads is higher. Similarly, when the issue rate is high, most loads will unlikely be critical; even those that are critical will likely have relatively low *tautness* (the number of cycles that can potentially be saved before another path becomes critical [32]).

Restricting the predictor's training and usage based on issue rate has two advantages. First, we do not need to mon-

itor loads during periods of sufficient performance. This can reduce the storage requirements of the CLPT because collecting data for fewer loads reduces the aliasing in the CLPT. Consider a program with two alternating function calls f(A) and f(B). Due to data-dependent control flow within the function, f(A) may have six loads that always hit in the cache while f(B) only has two loads that usually miss. Without filtering, these eight loads would need at least eight CLPT entries to store all of their criticality information. Since f(A)'s execution maintains a high instruction throughput due to the cache hits, criticality-based optimization of the loads in f(A) are unlikely to provide much benefit, and so storing the corresponding criticality predictions in the CLPT wastes space. If the CLPT updates only occur during periods of low performance, then updates from f(A) will be entirely filtered out, and only the two updates from f(B) will be stored in the CLPT. The result is that the CLPT only needs two entries to store this information. Another secondary advantage of filtering CCL/CLPT activity is that it may not be necessary to explicitly account for branch or path history when making criticality predictions. The two loads in f(B) may have the same PCs as two of the loads in f(A). If the predictors are always updated, then it may be necessary to include, for example, branch history to distinguish between the different instances of the loads. At least in this example, our filtering mechanism would simply eliminate the loads from f(A), eliminating the path-ambiguity from the predictor. To determine the critical period, we make use of the issue rate of the processor. Every cycle the number of instructions issued is monitored and every s cycles a global or average issue rate for the processor is noted. If the processor utilization falls below a pre-specified target issue rate, this implies that a significant portion of the processor resources are sitting idle waiting for one or more stalls to resolve. When the global issue rate falls below this target issue rate, we enable the tracking of load consumers and subsequent updates to the CLPT.

We also include a 2-bit confidence counter for each entry in the CLPT. This counter is incremented if the difference between the previous and current number of consumers is less than Δ=2, and decremented otherwise. In other words the counter is incremented if the number of children is reasonably stable, and decremented if the number varies too much. If this confidence counter is low, then we assume the load is critical regardless of the number of consumers stored in the CLPT entry. To maintain implementation simplicity, we do not repair the CLPT (or update consumer counts) on pipeline flushes, as the design overhead to repair consumer counts due to wrong-path consumers is non-trivial. This could (slightly) impact the accuracy of the criticality prediction, but our confidence counters can protect against these cases. For the results presented in this paper, we use the following parameters for our load criticality predictor. Our largest criticality threshold used is twelve, and therefore the child counters associated with each ROB entry each only use four bits. The exact threshold values were empirically chosen; for different microarchitectures or cache hierarchies, different thresholds may be used. The CLPT has 1024 entries, each of which consists of a four-bit field to store the observed number of children, and then also the two-bit confidence counter. We also include another one-bit field that will be described in Section 5. The total amount of state required for the CLPT is 1024 entries $\times$ 7 bits per entry = 896 bytes. Even when including the extra bit per RAT entry and the per-ROB entry 4-bit consumer count, the total state is still under 1KB. Our simulated processor has a peak execution width of six μops per cycle, and we only enable the predictor when the actual issue rate is less than four.

While previous studies have proposed predictors that are similar in spirit, the specifics of how to build hardware to collect the required information had been omitted; in many cases these predictors are perhaps easy to add to a simulator, but non-trivial to build a practical circuit for. We have provided a detailed explanation of the microarchitecture-level implementation of our load criticality predictor. The modifications are few (one extra bit per RAT entry without requiring any extra ports, the ROB counters which can be implemented as a separate table to avoid impacting ROB area and timing, the simple PC-indexed CLPT which can be easily accessed in the processor front-end off any critical timing paths, and a few bits in the RS or LDQ entries to record the actual criticality predictions), which leads us to conclude that the predictor is practical to implement.

3.4. Intuition for Predictor Effectiveness

Our predictor works on the assumption that when a load has a larger number of consumers, the probability that *at least* one of its consumers lies on a critical path increases. For every committed load, we tracked the number of consumers as well as the instruction types of each consumer. Figure 3 shows the breakdown/percentage of consumer instruction type for loads with varying numbers of consumers. We separate the instruction types into six categories: low latency (instructions like add, subtract, multiply and so on), cache hit (loads that hit in the DL1 cache), correct branch (correctly predicted branches), cache miss (loads that miss in the DL1 cache), mispredicted branch, and long latency (instructions like divide and other complex instructions). The first three categories can be regarded as instructions that will not experience very long latencies and are therefore unlikely to significantly impact processor performance. The last three categories are instructions with long latencies and are likely to be critical.

At first glance, it does not appear that many consumers are likely to be critical. Even in the cases where a load has twelve consumers, only about 20% fall into one of the likely-to-be-critical categories. This twelve-child load,

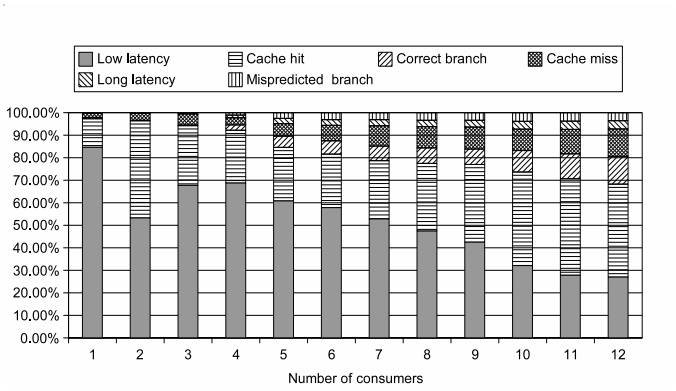

Figure 3: Distribution of consumers (for different consumer count values) according to instruction type.

however, only needs *one* of its children to be critical for itself to likely be critical. The probability of each consumer to be *not* critical is 80%, and so the probability of *all* twelve of the load's consumers not being critical is only 0.8^{12} = 6.9%. In practice, this probability may be even lower since we only consider the load's immediate consumers, whereas some of the low-latency operations may in turn have critical consumers. The observed consumer instruction-type characteristics combined with this statistical argument provide a simple and intuitive explanation for why our predictor is able to perform very well. Srinivasan et al. proposed a predictor that explicitly used an instruction's type to determine criticality. Our results suggest that a load's consumer count alone can implicitly lead to the same final conclusion about whether a load is likely to be critical or not.

4. Simulation Infrastructure

Our simulation infrastructure uses a cycle-level model built on top of a pre-release version of SimpleScalar for the x86 ISA [2]. This simulator models many low-level details of a modern x86 microarchitecture including a detailed fetch engine, decomposition of x86 macro instructions to micro-op (μop) flows, micro-op fusion, execution port binding [26], and detailed cache architectures including all inter-level buses, MSHRs, etc. Our baseline configuration is loosely based on the Intel Core 2 [9], employing a 96-entry reorder buffer, 32-entry issue queue, 32-entry load queue, 20-entry store queue, and a 14-stage minimum branch misprediction latency. The processor can fetch up to 16 bytes of instructions per cycle (properly handling x86 instructions that span cachelines as two separate fetches) using a 5KB hybrid branch predictor [20], is 4-wide throughout its in-order pipelines (decode/dispatch/commit), and 6-wide at issue, with three integer ALUs, an integer multiplier, one load port, one store port, one FP adder and one FP complex unit (some of these units are bound to the same issue ports, which is why the number of functional units exceeds

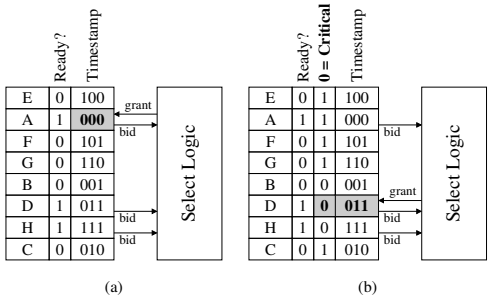

Figure 4: (a) Conventional timestamp-based oldest-first select logic, (b) augmenting the select logic to support criticality-prioritized select

the issue width). We employ a load-wait table based memory dependence predictor [17]. The processor has 32KB, 3-cycle, 8-way L1 caches, a 4MB, 16-way, 9-cycle L2 cache, and DL1 and L2 hardware "IP-based" prefetchers [9]. Main memory has a simulated latency of 250 cycles.

We make use of programs from the following suites: SPECcpu2000 and SPECcpu2006, MediaBench [13, 18], BioBench [1] and BioPerf [3]. All simulations warm the caches and branch predictors for 500 million instructions and then perform cycle-level simulation for 100 million instructions. We use the Sim-Point 3.2 toolset to choose representative samples [15]. Some applications do not yet run on SimpleScalar/x86 due to unsupported system calls and libraries.

5. Optimization Class 1: Faking the Performance of a Second Load Port

5.1. Problem Description

For modern out-of-order processors, a data cache with only a single read port may limit performance. Providing additional read ports is not simple because the area of SRAM arrays (such as the DL1 cache) increases quadratically with the port count. Besides the additional die area, the larger area increases the circuit path lengths which results in both higher latencies and higher power consumption. In an industrial simulator that we had access to, a second load data port increased the total active power of the processor by more than 13%. As such, all out-of-order Intel x86 processors since the Pentium-Pro have only supported a single load port [14, 16, 26]. We would like to have the performance benefits of a second load port, but we do not want to pay the area and power costs.

5.2. Implementation Details

Our first optimization uses load criticality to prioritize accesses to the single DL1 read port so that critical loads are given a higher priority. We use the consumer counter stored in the CLPT to determine how many immediate consumers are predicted to be waiting on a load. We also track whether a load was ready to issue but delayed due to contention

for the data cache read port (i.e., there was another ready, older load that the select logic chose to issue instead). We augment each entry of the CLPT with the extra "ready-but-delayed" bit.

Our *non-critical load deferring* technique works as follows. If the CLPT value for the load is below the criticality threshold (five for this optimization), and the "ready-but-delayed" bit is 0, the load is determined to be not critical. We then modify the select logic to give critical instructions higher priority. The select logic typically employs an oldest-first policy for choosing a ready instruction to issue. One way of implementing this is to have each instruction keep a timestamp, where the oldest instruction has the smallest timestamp [6], as shown in Figure 4(a). Instruction A is the oldest ready instruction (timestamp=000_2) and receives the grant to issue. To modify the select logic to account for criticality, we extend the timestamp by a single bit: critical instructions will have this bit set to zero, while all other instructions will have this bit set to one. Figure 4(b) shows the same example instructions with the extended timestamps. Notice for example, that while load D comes later in program order than A (ignoring the extended bit of the timestamp, D has a timestamp of 011_2 while A has a timestamp of 000_2), the select logic effectively believes that D is the "older" instruction because its extended timestamp has a smaller value (0011_2 for D, and 1000_2 for A). Since we do not actually allow multiple loads to issue to the cache per cycle, this means that we do not require any additional tag broadcast buses for the wakeup/scheduling logic, nor do we need any additional result bypass paths beyond the path already present for the original load port.

The second component that we consider, in this optimization, is an additional load address port (address generation unit or AGU). Note that load execution is effectively broken into two stages: first the load issues from the reservation stations to the AGU, which then deposits the address in the LDQ. The load may need to wait in the LDQ for some number of cycles until it is permitted to access the cache; this could occur due to predicted memory dependencies on earlier stores [21] or because the DL1 SRAM read port is busy. In this component, we propose to hijack the store AGU that currently only serves store address μops to also execute load addresses. The output of this AGU must now be routed to both the LDQ and the STQ, but this is far simpler than if we had to, for example, add another result bus to the bypass network. To prevent starvation of non-critical loads, we only allow a particular load to be deferred up to three times after which it gets priority to access the load port.

5.3. Performance Evaluation

We present the performance of this application of load criticality in two scenarios. First we incorporate non-critical load deferring in a processor configuration with a single DL1 read port and a single load address port and compare this to a processor with a single load address port and two *true* DL1 read ports. Figure 5 shows the relative speedup when critical load prediction is used to prioritize load port usage. We present averages across all of the benchmark suites as well as present per benchmark results for the SPEC2000 and SPEC2006 suites. FSLP (Faking a Second Load Port) corresponds to deferring non-critical loads and PSLP (Pure Second Load Port) shows the performance if a complete additional DL1 read port is added. As we can see, for almost all of the simulated applications, FSLP is able to match the performance of the PSLP. For a few applications like *crafty*, *eon* and *bzip2-06*, the performance of FSLP is slightly lower than PSLP; in these applications we observed many loads being predicted as critical and hence could not be deferred. For *mgrid*, FSLP slightly outperforms PSLP. In this application, we observed extreme contention for the load ports in the PSLP configuration. In this case, it is better to issue a single critical load rather than simply the two oldest loads which may be non-critical. For our simulations, the PSLP results are slightly idealistic because we have not accounted for the latency increase that adding a second read port would cause; in practice, our FSLP's performance would be closer to, if not better than, a real dual-read-port DL1 cache.

In the second part of this experiment we simulated the baseline with two load address ports. Providing two load AGUs *and* two DL1 read ports improves performance over the baseline by 14.8%. In the FSLP technique, an extra AGU can expose additional ready loads, both critical and non critical, for the LDQ to send to the data cache (i.e., there are more opportunities to reschedule loads based on criticality). As explained earlier, rather than adding another functional unit for the second load AGU, we use the same functional unit that is used to compute store addresses. We refer to this strategy as FSLA (Faking Second Load AGU). The third bar in Figure 5 shows the performance of FSLP augmented with the FSLA technique (FSLP+FSLA), and the fourth bar shows the performance benefit of having two complete load data ports and two complete load AGUs (PSLP+PSLA). The PSLP+PSLA technique here is really an ideal truly dual-ported scenario. For some benchmarks like *hmmer*, this difference is close to 4%, where the dedicated second load AGU prevents loads from competing with stores for the AGU. Additionally, when there are more ready loads available, the PSLP technique is able to issue two loads per cycle. Even though some of these issued loads may be non-critical, they may expose some other independent work that can keep the processor busy. On average, however, the performance of FSLP+FSLA is within 1.7% of PSLP+PSLA, which is not much when one considers the low cost of our FSLP+FSLA compared to the overhead of implementing a true second load AGU and DL1 read port.

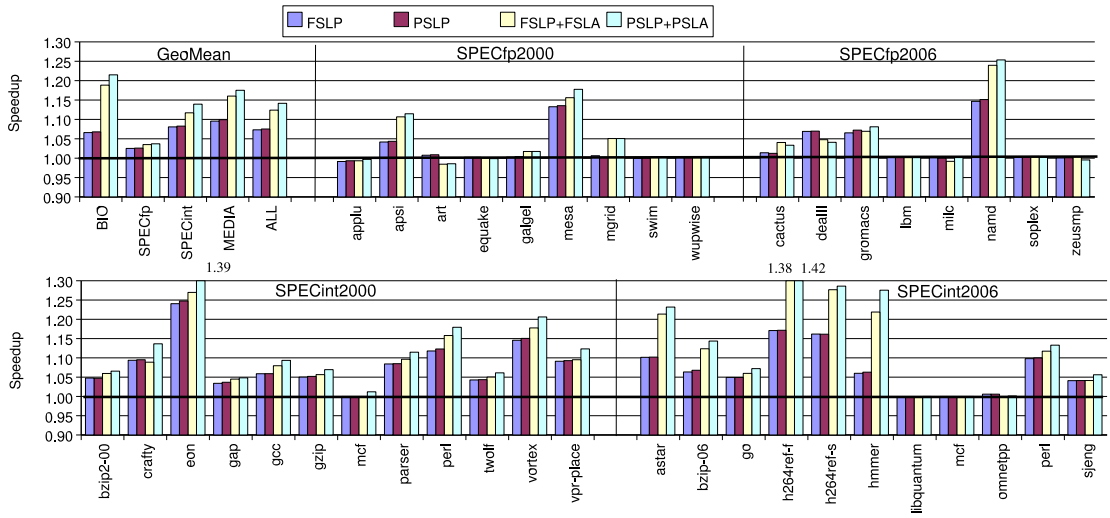

Figure 5: Deferring non-critical loads to achieve the performance of a second load data port.

5.4. Why Does FSLP Work?

The previous sub-section demonstrated the performance benefit that load criticality prediction provides, when used to prioritize access to the data cache port. To understand why FSLP performs so well, we conducted an experiment that measures the average latency in cycles from when loads are ready to issue (address computation and memory disambiguation are complete) to when they actually issue. A delay in load issue in this experiment corresponds to the number of cycles that a load was forced to wait for the DL1 port to become available.

Table 1 shows, for all of the application suites, the issue delay in cycles for critical and non-critical loads for the three load port configurations. The first column of the table indicates the name of the suite, and the second and third column give the average latencies for a single DL1 port for critical and non-critical loads, respectively. The issue delay values might seem slightly high since the average value is biased by certain applications which experience very long delays as well as some effects of contention in the MSHRs. The fourth and fifth columns represent a single DL1 port augmented with FSLP while the last two columns correspond to the PSLP configuration. There are two important results that can be obtained from this experiment. First, in all four simulated benchmark suites, the issue delay of critical loads in the FSLP configuration is nearly equal to that of the PSLP configuration. When a load is critical, having a true second load port allows the critical load to issue much sooner. With FSLP, critical loads experience a similar wait time. Second, since our FSLP approach still only has one real load port, the average ready-to-issue delay for non-critical loads is significantly higher than in the PSLP case. This has little impact on performance, however, since by definition these loads are not (likely to be) critical. These results explain why our FSLP approach performs nearly as well as the PSLP configuration.

Benchmark Suite	One Load Port		FSLP		PSLP	
	CL	NCL	CL	NCL	CL	NCL
BIO	14.8	16.4	3.4	14.4	3.3	6.6
SPECfp	12.1	16.2	4.4	11.5	4.4	8.5
SPECint	15.1	16.3	6.6	13.0	6.3	6.4
MEDIA	23.6	23.9	5.8	14.0	5.6	8.7

Table 1: Average ready-to-issue delay in cycles for critical (CL) and non-critical loads (NCL).

6. Optimization Class 2: Data Forwarding and Memory Disambiguation

6.1. Data Forwarding

Every load normally performs an associative search of the store queue (STQ) to determine if it should receive a forwarded value from an older store to the same address. This CAM-based search comprises a significant portion of the store queue's power consumption. To avoid performing these associative searches, one alternative is to force loads to wait until all earlier stores have committed and finished writing back to cache/memory. At this point, a load can safely issue to the data cache and be guaranteed to receive the most recent, architecturally correct value. Unfortunately, forcing all loads to wait for all earlier stores to commit and writeback can unnecessarily delay load execution. In our experiments, constraining loads this way caused an average performance degradation of 5% (but a 100% reduction in STQ searches). We observed, however, that while all loads associatively searched the STQ, on average only 13% of loads matched in the STQ and received forwarded data.

Using our load criticality predictor , we only allow the predicted critical loads to search the STQ. Only these loads can receive forwarded values from the STQ, while all other

non-critical loads must wait until all older stores have written back to the data cache. The idea is that the power cost of repeatedly searching the STQ far outweighs the small potential performance benefit that can be achieved from earlier execution of loads that are not likely to be critical in the first place. Note that this optimization needs to be conservative since the prevention of even a few critical loads from receiving forwarded data through the STQ can significantly hurt performance, so we set a low criticality threshold of two. We also monitor the global issue rate when deciding if a load should be prevented from searching the store queue. If the issue rate of the processor is very low (less than three μops per cycle on average), we allow all loads, irrespective of criticality, to search the store queue.

Applying this optimization of critical-load prediction reduces over 93% of the store queue searches with less than 1% performance loss on average. Only one benchmark, *art* belonging to the SPECfp2000 suite saw a 7% slowdown due to missed forwarding opportunities. This further reinforces the fact that most of the STQ searches result in no matches, which has been exploited by several other studies not directly related to load criticality [25, 29]. We extended our analysis to estimate the energy savings that this reduction in STQ searches would provide. We measured the activity numbers for the STQ in our simulator and used CACTI 4.1 to estimate the read, write and tag broadcast energy for our simulated STQ configuration [30]. Our results showed that reducing 93% of the store queue searches reduces the energy consumption of the STQ by 16%. Note that while this energy reduction does not account for the additional energy consumed by our predictor, we are not really explicitly proposing this STQ optimization to be implemented in *isolation*, but rather in conjunction with the other criticality-based techniques described in this paper. Once one has already paid the cost of the predictor to provide the performance of a second load port, then this STQ optimization can effectively be had for no additional cost.

6.2. Memory Disambiguation

Our next application in this class involves speculative disambiguation of memory instructions. Before a load can issue, it needs to check if there are any older stores in the instruction window that have not resolved their addresses. If the load finds any older store with an unknown address, then the load cannot safely issue since it may have a true data dependency with this store. Most experimental data has shown that the majority of loads do not have true dependencies with any stores currently in the processor [21]. To prevent all loads from incurring this stall, some processors use memory dependence predictors that, based on past load behavior, predict whether a load will collide with any earlier unknown stores [9, 17]. If the prediction is incorrect, then all instructions after the load get flushed from the pipeline and need to be refetched.

Past research has shown that accurate memory dependence prediction has the potential to increase performance [8], but these memory dependence predictors need to be fairly large to achieve maximum load coverage and accuracy. In this optimization, we only allow critical loads to access the memory dependence predictor and speculatively execute. All non-critical loads must wait until all previous store addresses have been resolved. The non-critical loads can usually afford to wait and resolve all dependence ambiguities since they will not have a significant impact on performance. There is no need to risk a misprediction and subsequent pipeline flush on a non-critical load where correctly speculating is unlikely to provide any significant performance benefits anyway. There is also less destructive interference in the memory dependence predictor since only critical loads update the predictor table. This allows a much smaller memory dependence predictor to be implemented while maintaining the same performance levels as the baseline predictor. In particular, using a memory dependence predictor modeled on a load-wait table predictor (like that used in the Alpha 21264 [17]), we were able to reduce the number of entries in the predictor table from 1024 to a meager 64 with no loss in performance.

7. Optimization Class 3: Data Cache

7.1. Insertion Policy for Non-Critical Loads

The first application in this class deals with the insertion policy used on a cache fill. Previous research has shown that inserting cache lines in the least-recently-used (LRU) position instead of the conventional most-recently-used (MRU) position can sometimes improve performance [22]. If the line is accessed a second time (cache hit), then it is promoted to the MRU position in the recency stack. The LRU insertion policy has the benefit of preventing lines with low reuse from residing in the cache for long periods of time. The drawback to this policy is that while it helps some applications that have very low reuse, it also hurts other applications with short or medium reuse distances. Adaptive selection of the insertion policy has been propose to deal with this issue [22].

Rather than use an adaptive control scheme, we simply use our load-criticality predictions to guide the insertion policy. Data brought in by non-critical loads get inserted in the LRU position in the cache, where they are more vulnerable to eviction. All data read by critical loads follow the baseline MRU insertion policy, where several cache accesses must occur before the line is in danger of eviction. This organization uses a criticality threshold of eight.

7.2. Cache Bypassing for Non-Critical Loads

Our second cache application deals with the placement of the data in the cache hierarchy. Some loads do not only have low reuse, but they sometimes have *no* reuse or extremely large reuse distances. In the first scenario, the data required

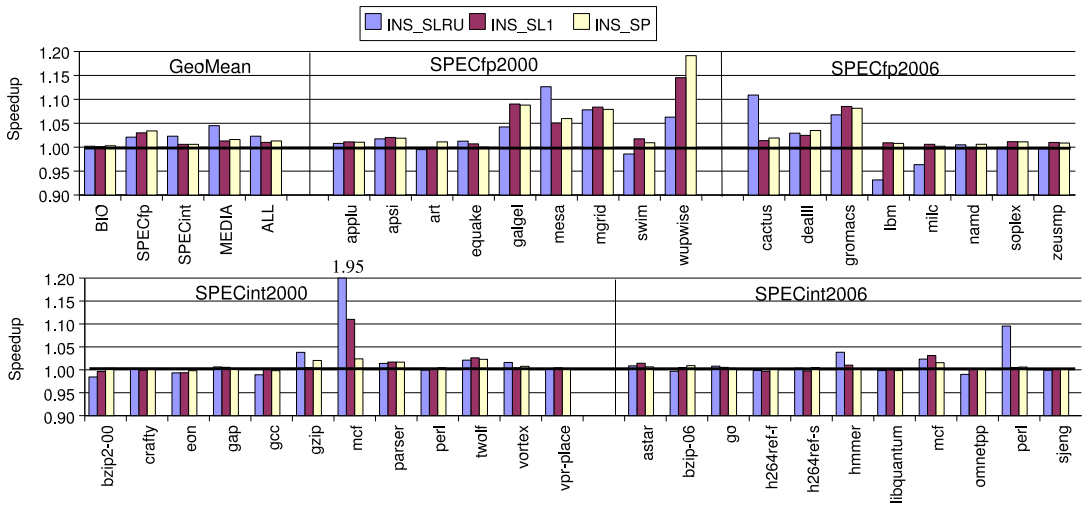

Figure 6: Speedup when critical load prediction is employed to optimize the L1 data cache.

by the applications are used only once and then sit idle in the cache until they are evicted. In the second case, the data is sure to get evicted from its set before it can be reused. In either case, these cache lines occupy space in the DL1 that could be better used by other data.

We use our load criticality information to simply prevent cache lines accessed by non-critical loads from being installed in the L1 data cache (i.e., they bypass the L1). Since this data is anyway only used by non-critical loads, we can afford the extra cycles required to access the L2 cache. Preventing non-critical loads from bringing data into the L1 reduces the cache pressure and can improve the hit rates for critical loads. As discussed in Section 1, while our non-critical load bypassing has similarities to other earlier proposals that prevent the data for non-critical loads from entering the DL1, our approach does not require building any additional caching structures such as the Non-Critical Load Buffer [12] of the Penalty Buffer [4]. We use a criticality threshold of four for this application.

7.3. Prefetching Policy for Non-Critical Loads

The third application deals with the DL1 hardware prefetcher. In our simulator, turning off data prefetching caused an average performance degradation of 2.9%. A prefetching algorithm is often difficult to tune and can end up polluting the cache by bringing in data that is never used. Prefetching can also cause additional contention for the off-chip bus, thereby causing further performance degradations. Unnecessary prefetches are even worse than the problems described for cache insertion and placement since, in those cases, the data that are brought in were used at least once. For this application we simply prevent non-critical loads from causing any prefetch requests. By restricting the prefetching algorithm to bring in data only for critical loads, cache pollution and bus utilization are both decreased. The criticality threshold for this technique is five.

7.4. Performance Evaluation

We now present results for the three applications described above. Figure 6 shows the performance speedup with respect to the baseline for the three optimizations. The first bar shows the performance due to inserting non-critical loads in the LRU position. While the overall performance improvement is 2.3%, some applications see large speedups. In particular, mcf sees a 95% improvement; mcf has low data reuse and experiences a 38% speedup even when the INS_LRU technique is applied. By limiting this strategy to only non-critical loads, the speedup is increased. Among the non-SPEC benchmarks suites, the media applications have nearly 5% performance improvement on average. Although not shown on this graph, *jpeg-encode* observes a 25% speedup. For this benchmark, the INS_SLRU technique reduces the miss rate of critical loads in the L1 cache by 10%. When compared to the INS_LRU technique which inserts all loads in the LRU position, this strategy degrades the performance of fewer applications. The second application in this class measures the performance benefit obtained by completely preventing non-critical data from entering the L1 data cache (INS_SL1 = Insert Selectively in L1). Figure 6 shows that this optimization achieves an average performance improvement of 3% for the SPECfp suite and 1% overall. While the average improvement is not very high, certain benchmarks do benefit. Finally, the third bar in Figure 6 (INS_SP = Insert Prefetched Data Selectively) shows the impact of criticality-based filtering of prefetches. The results indicate that the floating point benchmarks seem to benefit the most by this technique. The average performance improvement for the SPECfp suite is nearly 4%, with *wupwise* showing speedup of nearly 20%. For *wupwise*, only 0.1% of prefetches in

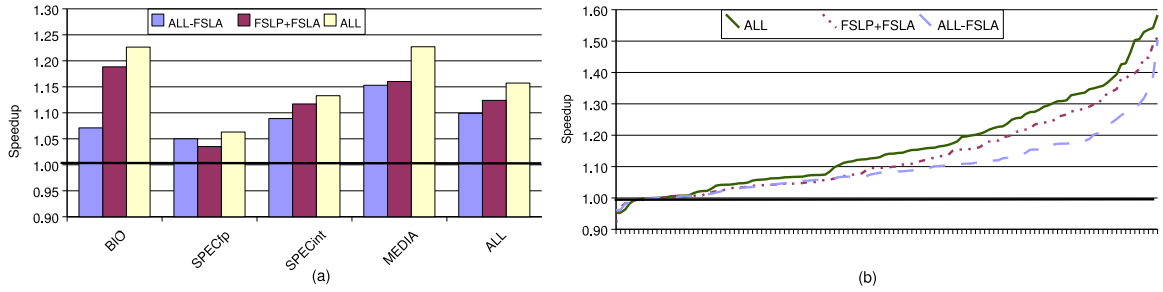

Figure 7: Speedup when combining all optimizations: (a) per-suite averages and (b) S-curves for all benchmarks.

the baseline configuration turned out to be useful, so when load criticality prediction is used to filter out even some of the useless prefetches, performance improves considerably. Even though each of these techniques only improves average performance by a few percent, we emphasize that these benefits are effectively free once the basic criticality predictor has been implemented.

8. Combining All Load Optimizations

In this section we study the impact of simultaneously combining our optimizations. We present three sets of optimization configurations in Figure 7: faking a second load data port with hijacking the store AGU (FSLP+FSLA), combining all of the optimizations (ALL), and combining all except for hijacking the store AGU (ALL-FSLA). The ALL-FSLA configuration is provided to quantify the impact of all of the criticality-based optimizations without the impact of hijacking the store AGU. Note that for the ALL configurations, selective insertion policy takes precedence over the DL1 bypass optimization since it provided higher performance (insertion position is irrelevant if the DL1 is bypassed).

Figure 7(a) shows the average results for the individual benchmark suites. Most of the suites see considerable speedups even with the ALL-FSLA combination.The FSLP+FSLA technique provided a speedup of 12.3% on average, while the ALL combination achieves a speedup of 15.7%. Figure 7(b) shows the S-curves. When combining all optimizations (ALL), we see that the majority of benchmarks experience 10% or more improvement, a few benchmarks are performance neutral, and only a very small number exhibit some performance degradation. Recall that some of our optimizations target the reduction of power and the size of related hardware structures. For example, *art* (SPECfp2000) experienced a 7.0% performance degradation due to the fact that filtering non-critical loads from searching the STQ causes this benchmark to miss too many forwarding opportunities. When combining all of the optimizations, however, this degradation reduces to 3.5%. One can reduce this performance degradation even further by adjusting the thresholds to be more conservative about when to block loads from searching the STQ.

While our criticality predictor is certainly not the first one to be proposed, we believe that it is simpler and more practical to implement than several of the earlier predictors. We simulated our criticality-based load optimizations (the simultaneous combination of all techniques on our academic simulator) using Tune et al.'s QCons heuristic that in each cycle finds the completed instruction with the largest number of children [31]. Note that in Tune et al.'s study, they used a 64K-entry predictor, which we model here, but we keep our own consumer-based predictor sized at only 1K entries. Due to space constraints we omit report per-benchmark numbers. Our simulations indicate that the QCons heuristic is a fairly good indicator of load criticality and it provides a 12.3% average speedup as compared to the 15.7% speedup provided by our predictor. Tune et al.'s predictor, however, uses 64× more entries than our predictor, which shows that our predictor might be a more viable and efficient option.

9. Conclusions

While load criticality has been extensively studied for optimizing value prediction and clustered microarchitectures, we have demonstrated that the idea is useful even for conventional, main-stream processor microarchitectures. We have been able to use *and reuse* a single, simple criticality predictor to optimize multiple facets of load execution. Our predictor, along with our optimizations that use the predictor, have all been designed with an eye toward minimizing microarchitectural impact. As such, we believe that these techniques are indeed practical to implement in near-term future microarchitectures.

An interesting direction for future research is to explore and understand how instruction criticality interacts with cross-core interference in a multi-core processor. This may require new fundamental definitions of criticality that account for interactions between the cores (is an instruction critical if delaying hurts one core's performance but improves another's due to the change in the order that the cache hierarchy receives memory requests?). Once such definitions have been established, additional research will be needed to develop mechanisms to exploit this information to better optimize the performance and/or power of multi-core processors. In any case, we have demonstrated

that instruction criticality is useful for traditional microarchitectures, and we believe that there are still many more opportunities to exploit it.

References

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 2–9, Austin, TX, USA, March 2005.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.

[3] D. A. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture of Bioinformatics Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 163–173, Austin, TX, USA, October 2005.

[4] R. I. Bahar, G. Albera, and S. Manne. Power and Performance Tradeoffs Using Various Caching Strategies. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 64–69, Monterey, CA, USA, August 1998.

[5] A. Baniasadi and A. Moshovos. Asymmetic-Frequency Clustering: A Power-Aware Back-end for High-Performance Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 255–258, Monterey, CA, USA, August 2002.

[6] A. Buyuktosunoglu, A. El-Moursy, and D. Albonesi. An Oldest-First Selection Logic Implementation for Non-Compacting Issue Queues. In *Proceedings of the 15th International ASIC Conference*, pages 31–35, Rochester, NY, USA, September 2002.

[7] B. Calder, G. Reinmann, and D. Tullsen. Selective Value Prediction. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 64–74, Atlanta, GA, USA, June 1999.

[8] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.

[9] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. White paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[10] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing Performance Under Technological Constraints. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 47–58, Anchorage, AK, USA, May 2002.

[11] B. Fields, S. Rubin, and R. Bodík. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 74–85, Göteborg, Sweden, June 2001.

[12] B. R. Fisk and R. I. Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve data Cache Efficiency. In *Proceedings of the International Conference on Computer Design*, pages 538–545, Austin, TX, USA, October 1999.

[13] J. E. Fritts, F. W. Steiling, and J. A. Tucek. MediaBench II Video: Expediting the Next Generation of Video Systems Research. *Embedded Processors for Multimedia and Communications II, Proceedings of the SPIE*, 5683:79–93, March 2005.

[14] S. Gochman, R. Ronen, I. Anati, A. Berkovitz, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(2), May 2003.

[15] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, Madison, WI, USA, June 2005.

[16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyler, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[17] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro Magazine*, 19(2):24–36, March–April 1999.

[18] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, USA, December 1997.

[19] D. Marculescu. Application Adaptive Energy Efficient Clustered Architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 338–343, Newport Beach, CA, USA, August 2004.

[20] S. McFarling. Branch Predictor with Serially Connected Predictor Stages for Improving Branch Prediction Accuracy. USPTO Patent 6,374,349, April 2002.

[21] A. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin, 1998.

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer. Adaptive Insertion Policies for High-Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, San Diego, CA, USA, June 2007.

[23] P. Salverda and C. B. Zilles. A Criticality Analysis of Clustering in Superscalar Processors. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 55–66, Barcelona, Spain, November 2005.

[24] J. S. Seng, E. Tune, and D. M. Tullsen. Reducing Power with Dynamic Critical Path Information. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 114–123, Austin, TX, USA, December 2001.

[25] T. Sha, M. M. K. Martin, and A. Roth. NoSQ: Store-Load Forwarding without a Store Queue. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 285–296, Orlando, FL, December 2006.

[26] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* McGraw Hill, 2005.

[27] A. R. L. Srikanth T. Srinivasan, Roy Dz-ching Ju and C. Wilkerson. Locality vs. Criticality. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 47–58, Göteborg, Sweden, June 2001.

[28] S. T. Srinivasan and A. R. Lebeck. Load Latency Tolerance in Dynamically Scheduler Processors. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 148–159, Dallas, TX, USA, November 1998.

[29] S. Subramaniam and G. H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue At All. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 273–284, Orlando, FL, December 2006.

[30] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories Palo Alto, June 2006.

[31] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 185–196, Monterrey, Mexico, January 2001.

[32] E. S. Tune, D. M. Tullsen, and B. Calder. Quantifying Instruction Criticality. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.